Spring
‘24

Games Rating Model

MILESTONE 1

CS TEAM 9

Name	ID
Yusuf Nasser Saber Blattes	20191700794
Ayman Mohamed Ahmed Hasan	20201700168
Samy Samer Fayez Hanna	20201700340
Maria Sameh Makram Boles	20201700633
Fatma Awad Ismail Abdullah	20201700570
Youssef Ahmed Abdellatif Gad	20201700995

Preprocessing Techniques

In the realm of machine learning, especially in regression and classification models that don't use neural networks and deep learning, data quality is paramount. While having a large amount of good data is important, it's only part of the equation. Preprocessing is the task of extracting the best possible features from that data to achieve optimal results. In this section, we will delve into various preprocessing techniques and how our team approached this crucial step in our project.

1. Release Date, Last Update Date

To ensure accurate loading of dates before preprocessing, our first task was to determine the date format. Through data exploration, we discovered that the dates were in the format 'day/month/year'. With this information, we developed a datetime parser to allow the `read_csv()` function to correctly parse the dates in the dataset. This step was crucial in ensuring that the data was properly prepared for subsequent preprocessing techniques.

Having converted the dates to datetime format, we can easily convert them to integers via the `toordinal()` function.

We then proceeded to perform feature engineering on the dates columns, we extracted game age, time since last update, maintenance period features out of it as shown in the following code snippet:

```
# Create a new column with the age of the game
_df['game_age'] = datetime.now().toordinal() - _df['Original Release Date']

# Create a new column with the time since the last update
_df['last_update'] = datetime.now().toordinal() - _df['Current Version Release Date']

# Create a new column with the maintenance period
_df['maintaning_period'] = _df['game_age'] - _df['last_update']
```

Missing Data Filling: used `sklearn.impute.SimpleImputer` to impute the missing dates via the median strategy, fitted during training time.

2. In-app Purchases

To clean up the In-app purchases column, we first converted it to a string, removed brackets and quotation marks, and split the string on commas to create a list of individual purchases. We then iterated through each purchase value in the list and converted it to a float type.

With the purchases now in a cleaner format, we performed feature engineering to extract additional information. Specifically, we created new features to capture the number of in-app purchases for each app, including the highest purchase, lowest purchase, and average purchase.

Missing Data Filling: To ensure consistency in our analysis, we filled in any apps that didn't offer in-app purchases with **zeros** for all these new columns.

3. Age Rating

To clean up the age rating column, we first converted it to a string, removed the '+' sign and converted it to a float type, the column now indicates the minimum age allowed to download and play this game.

Missing Data Filling: used `sklearn.impute.SimpleImputer` to impute the missing ages via the median strategy, fitted during training time.

4. Developer

This column was one of the most challenging to work with, having only the developer's name is not all that helpful, we couldn't conduct full-blown research on the industry as a whole and figure out a way to evaluate each developer and boil it out to something like a popularity score, so our options were quite limited, but this didn't stop us.

After all, it was just a categorical column, we chose two approaches to apply here so we can extract a couple of features out of this column.

a. Target Encoding:

We first convert the *Developer* column to string format and remove any brackets or quotation marks. It then groups the data by developers and replaces the names of developers with less than two games with 'Other' whose value is reassigned an `np.nan`, later imputed when doing data filling.

We then proceed to use `category_encoders.TargetEncoder` to do the target encoding with a smoothing parameter of 20 to help add more generalization to the feature and decrease the chances of overfitting.

Missing Data Filling: used `sklearn.impute.KNNImputer` to impute the missing values the mean of the nearest 5 neighbors.

b. Frequency Encoding:

We also convert the *Developer* column to string format and remove any brackets or quotation marks.

We then proceed to use `category_encoders.CountEncoder` to do the simple count encoding.

Missing Data Filling: used `CountEncoder` instance to impute the missing values on its own via the default parameters.

5. Genres, Languages

The Genres and Languages columns posed a challenge during preprocessing since they contained a list of genres and languages rather than one. We attempted to use built-in preprocessing functions from the scikit-learn library such as `OneHotEncoder()` and `MultiLabelBinarizer()`, but they generated duplicate rows that were difficult to work with.

Eventually, we settled on using the pandas `get_dummies()` function for the dummy variable approach. This method involved removing certain genres ('Games', 'Strategy', 'Entertainment') from the list of genres for each observation and replacing genres with counts less than 2% of the total count with 'infrequent'.

Similarly, for the languages, we eliminated the 'EN' language as it was present in all rows and set a threshold of 10% of the samples that a language needed to appear in so it wouldn't be labeled as 'Infrequent'.

We then created dummy variables for each genre using the `get_dummies()` function and concatenated them to the original DataFrame. The NaN values in the dummy variables were filled with 0, and the original columns were dropped.

Another approach we tried was the NLP approach, where we first converted the 'Genres' column to a list of strings, dropped certain values as before, and joined the list of values into a single string. We then applied the `CountVectorizer` function to create a bag of words and used PCA to reduce the dimensionality of the bag of words. The PCA-transformed genres were then added to the original DataFrame, and the original column was dropped.

Overall, we found that cleaning these columns involved converting it to a list of strings, dropping certain values, and joining the list into a single string. From there, we used either the NLP approach or one of the dummy variable approaches to prepare the data for machine learning algorithms that require numerical inputs and reduced dimensionality.

6. User Rating Count, Price, and Size

These columns were read as floats value directly via `read_csv()` , no cleaning nor preprocessing needed.

7. Description

The 'Description' column was a challenging feature to preprocess because it contained descriptions provided by the game developers, who are likely biased towards their game and want to attract users to download and play their game. This meant that the descriptions were not necessarily objective and may contain marketing language that could skew our analysis.

To address this challenge, we first performed cleaning on the 'Description' column. This included removing any unnecessary characters or words that could potentially interfere with our sentiment analysis or other NLP techniques. We also removed any special characters that could impact the readability of the text.

After cleaning the data, we explored various NLP techniques to preprocess the 'Description' column. One approach we tried was to use topic modeling to identify the main topics or themes present in the descriptions. But it was pretty much moot as we already had the genres of the games.

Another approach we tried was to use sentiment analysis to determine the overall sentiment of the descriptions. We used the `nltk's SentimentIntensityAnalyzer()` to calculate a score. This allowed us to gain insights into how the descriptions might influence user ratings.

In addition to sentiment analysis, we also tried to compute an excitement score and an attractive score for each description using NLP techniques. The excitement score was computed as the sum of the positive and absolute negative polarity scores, while the attractive score was computed as the ratio of attractive keywords to total words. These scores helped us to identify which aspects of the descriptions were likely to be most appealing to users.

Overall, the 'Description' column posed a challenge due to potential biases and marketing language. Despite this challenge, we explored various NLP techniques to preprocess the data and gain insights into the descriptions' content and sentiments.

8. URL

At first glance, the URL column appeared to be of little value, as it simply contained a link to the game. However, upon further discussion, we realized that users visit these pages to rate the game, so there may be factors on the page that influence their rating.

To investigate this possibility, we decided to perform web scraping on the game pages and extract the reviews from them. By analyzing the reviews left by other users, we hoped to gain insights into the factors that may influence a user's rating.

Overall, this approach allowed us to gather additional data that could potentially be used as input variables for our regression models. By incorporating insights from the reviews into our analysis, we aimed to improve our understanding of the factors that influence user ratings and ultimately improve the accuracy and predictive power of our models.

To download all reviews from the website for each game, we used web scraping with Beautiful Soup 4. However, we discovered that 2400 games were no longer available on the website.

We elected to create two separate models, one with the reviews input and one without it.

9. Reviews

After downloading all the reviews from the website, we cleaned and preprocessed the data as follows: we compiled all the reviews for each game into a single string and then preprocessed the text by removing emails, numbers, punctuation, and stop words. This preprocessing step helped to ensure that the sentiment analysis was focused on the most relevant and informative text.

For the sentiment analysis itself, we used the VADER (Valence Aware Dictionary and sEntiment Reasoner) model. We chose this model because it is an online model that can be used without access to a GPU, which was a requirement for our analysis. The VADER model calculates a compound score that considers the positive, negative, and neutral sentiment expressed in the text.

By analyzing the compound score from the VADER model, we were able to determine the overall sentiment of the reviews for each game. By incorporating these sentiment scores into our analysis, we aimed to gain insights into the factors that influence user ratings and ultimately improve the accuracy and predictive power of our models.

that influence user ratings and ultimately improve the accuracy and predictive power of our models.

Missing Data Filling: used `sklearn.impute.KNNImputer` to impute the missing values the mean of the nearest 5 neighbors.

10. Icon

After downloading each icon, we first did object detection to count the number of objects in the icon via functions that converted the image to grayscale, then did edge detection via the canny edge detection technique, then proceeded to convert the edge map to an inverted binary image and finally applied cv2.findContours to return the number of objects detected.

Secondly, we processed them using a function that takes the file path of an image as input and returns a normalized feature vector. The preprocessing steps involved resizing the image to 100x100 pixels using OpenCV's `cv2.resize` function.

Next, we extracted color features using color histograms and shape features using edge detection. The color features were extracted using OpenCV's `cv2.calcHist` function and concatenated into a single feature vector. The shape features were extracted using the Canny edge detection algorithm and flattened into a 1D array. The color and shape features were then concatenated into a single feature vector.

To ensure that the feature vector was on a uniform scale, we normalized it to have unit length using NumPy's `np.linalg.norm` function. This normalization step was important for ensuring that the features were equally weighted and did not introduce any bias into our analysis.

Overall, this process allowed us to extract meaningful features from the app icons that could be used as input variables for our regression models. By incorporating these features into our analysis, we aimed to improve the accuracy and predictive power of our models.

11. Name, Subtitle

Name and subtitle are two columns that initially posed a challenge in terms of their relevance to our regression models. After much discussion, we considered dropping both columns altogether. However, we ultimately decided to leverage the potential value of these features by conducting sentiment analysis on the textual data contained within each. By extracting meaningful insights from these columns, we aimed to improve the accuracy and predictive power of our models.

12. Primary Genre

Out of 5200+ games, the primary genre for 5000 of them was ‘Game’. We decided it wouldn’t do any good to include it in the model as 98% of the column was representing the same value.

13. ID

We made the decision to drop the 'ID' column from our dataset. While this column served as a unique primary key during the preprocessing phase, we believed that including it as a feature in our regression models could potentially introduce noise and negatively impact performance. Therefore, we opted to exclude it from our final analysis.

Data Analysis

We performed several analyses on the data, right after we loaded it, while cleaning it and after finishing our preprocessing.

1. Exploration

a. Checking NaN values

```
[1] df.isnull().sum()
```

b. Exploring info

```
[1] df.head()
[2] df.info()
```

2. General Analysis

We analyzed 'Genres', 'Languages', 'Developer', and Dates columns to gain more clarity on what we should do about it them:

a. Genres

The following code snippet analyzes the values of the genres column by counting them to know which genres are the more frequent ones.

```
def genres_analysis(_df):

    _df['Genres'] = _df['Genres'].astype(str)
    _df['Genres'] = _df['Genres'].str.strip('[]').str.replace("'", "").str.split(", ")

    genre_counts =
_df.explode('Genres').groupby('Genres').size().sort_values(ascending=False)
    print(genre_counts)
```

The output of this analysis guided us to make a threshold so we don't have to include all the genres available but only a few of them, we keep the more frequent ones and exclude the others as infrequent.

b. Languages

The following code snippet analyzes the values of the Languages column by counting them to know which languages are the more frequent ones besides the first one which is the 'EN' language.

```
def lang_analysis(_df):
    _df['Languages'] = _df['Languages'].astype(str)
    _df['Languages'] = _df['Languages'].str.strip('[]')
                                       .str.replace("'", "").str.split(", ")

    langs_counts =
_df.explode('Languages').groupby('Languages').size().sort_values(ascending=False)
    print(langs_counts[1:30])
```

The output of this analysis guided us to make a threshold, so we don't have to include all the languages available but only a few of them, we keep the more frequent ones and exclude the others as infrequent.

c. Developer

The following code snippet analyzes the values of the developer column to identify how many unique developers are in the dataset and how many of them has more than one game published.

```
def dev_analysis(_df):

    print(_df['Developer'].value_counts())

    # print the number developers with more than 1 game
    print(len(_df['Developer'].value_counts()[_df['Developer'].value_counts() > 1]))

    print(_df['Developer'].unique().size)
```

The output of this analysis guided us to make a threshold, so we don't have to include all the developers available but only those who developed more than one game and exclude the others from the database as outliers.

d. Histogram Analysis

```
[1] df.hist(figsize=(15, 15))
```

e. Dates

The following function plots several histograms and boxplots on the dates columns and the features extracted from them to gain clarity on the similarities between them and identify whether or not the dates outliers are worth keeping, eventually we concluded that the outliers weren't errors nor anomalies and decided on keeping them.

```
def date_analysis(_df):
    # Plot the distribution of the date columns

    fig, ax = plt.subplots(5, 2, figsize=(20, 20))

    # df = date_preprocessing(df)

    # game_age distribution
    sns.histplot(_df['game_age'], ax=ax[0, 0])
    sns.boxplot(_df['game_age'], ax=ax[0, 1], orient='h')

    # last_update distribution
    sns.histplot(_df['last_update'], ax=ax[1, 0])
    sns.boxplot(_df['last_update'], ax=ax[1, 1], orient='h')

    # Original Release Date distribution
    sns.histplot(_df['Original Release Date'], ax=ax[2, 0])
    sns.boxplot(_df['Original Release Date'], ax=ax[2, 1], orient='h')

    # Current Version Release Date distribution
    sns.histplot(_df['Current Version Release Date'], ax=ax[3, 0])
    sns.boxplot(_df['Current Version Release Date'], ax=ax[3, 1], orient='h')

    # maintaning_period distribution
    sns.histplot(_df['maintaning_period'], ax=ax[4, 0])
    sns.boxplot(_df['maintaning_period'], ax=ax[4, 1], orient='h')

    plt.show()
```

3. Features Relations

During our feature selection process, we found that most of the features did not have a significant correlation with each other. This is a good thing as it allows for more variance in the data, which can improve the accuracy of our models. However, we did identify a few features that were highly correlated, specifically the languages dummy variables features and the dates features.

In the case of the languages dummy variables, we found that their correlation with the target variable was too weak to be included in our models anyway, so the fact that they were highly correlated with each other was not a problem.

For the dates features, we noticed that the features generated by subtracting today's date from the original columns were basically the same as the original features but reversed. Because the engineered features were more intuitive and easier to interpret, we elected to choose the engineered features and drop the original ones.

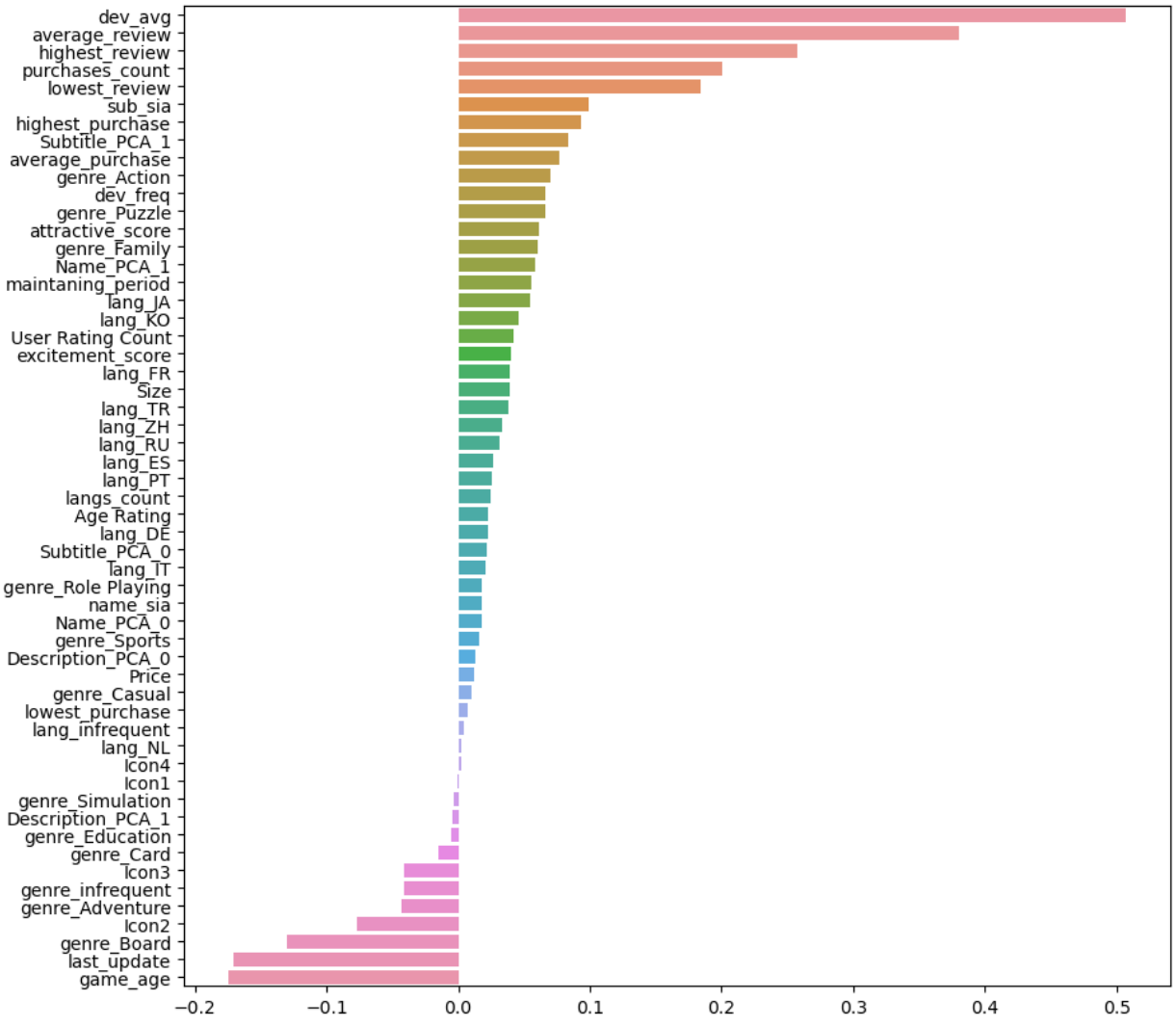
Overall, our feature selection process allowed us to identify the most relevant and useful features for our models, while avoiding issues such as multicollinearity or confounding variables. By carefully selecting our features, we can build more accurate and meaningful regression models that can help us to understand and predict user ratings for mobile games.

4. Correlation Analysis

Correlation analysis helps us to identify which features are most strongly related to the target variable in a regression model. This information can be used to select the best features for our models and to validate our expectations of the data.

Also allows us to identify potential issues such as multicollinearity or confounding variables that may affect the performance of our models. By interpreting the results of correlation analysis carefully, we can make informed decisions about which features to include in our models and how to interpret the results.

Overall, correlation analysis is an important step in the model building process, and can help us to build more accurate and meaningful regression models.



Regression & Classification Model

During our analysis, we used several regression techniques to identify the best methods for predicting user ratings for mobile games. While we tested many methods, two techniques consistently achieved the lowest Mean Squared Error (MSE) scores and highest R2 scores across multiple experiments.

1. XGBoosting

Is a powerful gradient boosting algorithm that is widely used in machine learning for both regression and classification problems. It is a type of ensemble learning method that trains a series of decision trees and combines their predictions to make a final prediction.

One of the key advantages of XGBoost over other gradient boosting algorithms is its ability to handle missing data and other types of data preprocessing. It also includes several regularization techniques to prevent overfitting and improve model performance.

Hyperparameter Tuning:

a. `max_depth`: This hyperparameter controls the maximum depth of each decision tree in the ensemble. Increasing max_depth allows the trees to capture more complex interactions in the data, but also increases the risk of overfitting. Decreasing max_depth can help prevent overfitting, but may result in lower model performance.

b. `n_estimators`: This hyperparameter controls the number of decision trees in the ensemble. Increasing n_estimators generally improves model performance, but also increases the risk of overfitting and can make training the model slower. Decreasing n_estimators can help prevent overfitting and make training faster, but may result in lower model performance.

c. `learning_rate`: This hyperparameter controls the step size at each iteration of the gradient boosting algorithm. A smaller learning_rate makes the model training slower but can improve the robustness of the model by allowing for finer adjustments to the weights of the decision trees. Increasing learning_rate can speed up training but may also increase the risk of overfitting.

2. CatBoost

Is gradient boosting algorithm that is designed to handle categorical variables in a more efficient way than other gradient boosting algorithms. It uses a combination of ordered boosting, gradient-based one-hot encoding, and random permutations of the categorical variables to handle categorical data.

One of the key advantages of CatBoost over other gradient boosting algorithms is its ability to handle missing data and categorical variables without the need for heavy data preprocessing. It also includes several regularization techniques to prevent overfitting and improve model performance.

Hyperparameter Tuning:

a. `max_depth`: This hyperparameter controls the maximum depth of each decision tree in the ensemble. Increasing max_depth allows the trees to capture more complex interactions in the data, but also increases the risk of overfitting. Decreasing max_depth can help prevent overfitting, but may result in lower model performance.

b. `iterations`: This hyperparameter controls the number of decision trees in the ensemble. Increasing `iterations` generally improves model performance, but also increases the risk of overfitting and can make training the model slower. Decreasing `iterations` can help prevent overfitting and make training faster, but may result in lower model performance.

c. `learning_rate`: This hyperparameter controls the step size at each iteration of the gradient boosting algorithm. A smaller learning_rate makes the model training slower but can improve the robustness of the model by allowing for finer adjustments to the weights of the decision trees. Increasing learning_rate can speed up training but may also increase the risk of overfitting.

d. `l2_leaf_reg`: This hyperparameter controls the L2 regularization term for the leaf weights of the decision trees. Increasing `l2_leaf_reg` can help prevent overfitting and improve model performance, but may also result in lower model complexity and underfitting.

3. RandomForest

Is a decision tree-based ensemble learning algorithm that combines multiple decision trees to make a final prediction. It works by building multiple decision trees on random subsets of the training data and features, and then averaging their predictions to make a final prediction.

One of the key advantages of Random Forest over other decision tree-based algorithms is its ability to handle high-dimensional and noisy data. It also includes several regularization techniques to prevent overfitting and improve model performance.

Hyperparameter Tuning:

a. `max_depth`: This hyperparameter controls the maximum depth of each decision tree in the ensemble. Increasing max_depth allows the trees to capture more complex interactions in the data, but also increases the risk of overfitting. Decreasing max_depth can help prevent overfitting, but may result in lower model performance.

b. `max_features`: This hyperparameter controls the maximum number of features that are considered for each split in the decision trees. Increasing max_features can improve the diversity of the trees and the overall performance of the model, but may also increase the risk of overfitting. Decreasing max_features can help prevent overfitting, but may result in lower model performance.

c. `n_estimators`: This hyperparameter controls the number of decision trees in the ensemble. Increasing `n_estimators` generally improves model performance, but also increases the risk of overfitting and can make training the model slower. Decreasing `n_estimators` can help prevent overfitting and make training faster, but may result in lower model performance.

d. `min_samples_split`: This hyperparameter controls the minimum number of samples required to split an internal node. Increasing min_samples_split can help prevent overfitting and improve model performance, but may also result in lower model complexity and underfitting. Decreasing min_samples_split can increase the complexity of the model and improve its ability to capture complex interactions in the data, but may also increase the risk of overfitting.

Training & Testing Splits

We used splits of 80% training and 20% testing to develop the model while also shuffling the data.

Screenshots of the resultant(s) regression line plots

Due to the large number of features, it would require working with high dimensionality, which is not feasible for creating plots.

Feature Selection

We dropped 'Primary Genre', 'ID' and completely discarded them regarding the features itself, we also removed both 'Icon URL' and 'URL' after downloading their contents and extracting the mentioned features before.

The feature selection part was done via `sklearn's SelectKBest` as follows:

```
# Feature selection
from sklearn.feature_selection import SelectKBest, f_regression
selector = SelectKBest(f_regression, k=10)
df_x_select =  selector.fit_transform(df_x, df_y)
```

SelectKBest is a feature selection technique in scikit-learn that can be used for both classification and regression problems. It works by selecting the K best features based on their scores from a univariate statistical test. The specific test used depends on the type of problem being addressed:

a. `f_classif`: This test is used for classification problems and computes the ANOVA F-value between each feature and the target variable. It measures the degree of linear dependency between the feature and the target.

b. `f_regression`: This test is used for regression problems and computes the F-value between each feature and the target variable. It measures the degree of linear dependency between the feature and the target.

In both cases, higher scores indicate stronger linear dependencies between the feature and the target variable. SelectKBest then selects the top K features with the highest scores and discards the rest.

Using SelectKBest with either `f_classif` or `f_regression` can help to improve the accuracy and interpretability of classification and regression models, respectively. By selecting only, the most relevant features, SelectKBest can help to reduce overfitting and improve the generalization of the model to new data.

Overall, SelectKBest is a powerful and flexible feature selection technique in scikit-learn that can be used with a variety of regression and classification models. It can help to improve model performance, reduce overfitting, and increase the interpretability of the model by selecting only the most relevant features.
The selected features (Not in this order):

1. game_age
2. last_update
3. dev_avg
4. genre_Board
5. purchases_count
6. highest_purchase
7. sub_sia
8. lowest_review
9. highest_review
10. average_review

Result Improving Techniques

We tried many approaches to improve the results but unfortunately most of them came out short, what we tried:

1. Eliminating outliers and Eliminating some features
2. Cross Validation
3. Regularization models
4. Multiple preprocessing techniques for the same column.
5. Hyperparameter Tuning

Regression Results

Model	Training MSE	Training R2	Testing MSE	Testing R2	Time Taken
Linear Regression	0.27	0.36	0.34	0.24	1.1 S
Ridge Regression	0.27	0.36	0.34	0.24	1.1 S
Lasso Regression	0.27	0.36	0.34	0.24	1.1 S
ElasticNet Regression	0.27	0.36	0.34	0.24	1.1 S
Polynomial Regression	0.25	0.41	0.33	0.26	5.8 S
XGBoost	0.20	0.52	0.32	0.27	2.6 S
Gradient Boosting	0.19	0.54	0.32	0.26	2.1 S
Random Forest	0.23	0.45	0.33	0.25	2.1 S
CatBoost Regression	0.13	0.67	0.32	0.29	3 S
AdaBoost Regression	0.27	0.35	0.35	0.21	2 S

Classification Results

Model	Training Accuracy	Testing Accuracy	Time Taken
Random Forest	75%	69%	0.8 S
CatBoost	74%	70%	1.2 S
XGBoosting	73%	67%	0.8 S
SVC	71%	67%	0.4 S

Thoughts and Remarks

The project involved a thorough exploration of various machine learning techniques and approaches, with a particular emphasis on the preprocessing of the data. Adhering to best practices, the training and testing phases of the project were kept strictly separate, with careful use of the pickle library to store fitted models, scalers, imputers, and encoders for later use in the testing phase. Throughout the project, data leakage was carefully avoided.

To ensure the data was properly prepared for modeling, several preprocessing steps were taken, including the downloading of icons and reviews, as well as the cleaning of the reviews prior to splitting the data. For classification tasks, the target variable was preprocessed to map its values to numbers ranging from 0 to 2, which did not affect the legitimacy of the data, nor cause any data leakage.

```
def rate_preprocess(_df):
  # Define mapping dictionary
  mapping = {
    'Low': 0,
    'Intermediate': 1,
    'High': 2}

  # Apply mapping to column
  _df['Rate'] = _df['Rate'].apply(lambda x: mapping[x])

  return _df

df_origin = rate_preprocess(df_origin)
```

Conclusion

As a team working on this machine learning model, we have concluded that data is a critical component in creating a successful model. To build a robust and accurate model, we need a large amount of relevant data that helps us to solve the problem at hand. Our feature selection process aimed to select the most relevant features for predicting user ratings for mobile games, based on what users might consider when rating a game, such as gameplay, performance, community engagement, content updates, and more.

While the data size collected was reasonable given the number of games available, we believe that the data gathered could have been more relevant to provide the model with more useful information. However, the features we selected, including Average Review, Highest Review, Lowest Review, Purchases Count, Highest Purchase, Game Age, and Last Update, all support our argument.

Upon analyzing the features selected for our machine learning model, we have found that many of them are highly relevant to predicting user ratings for mobile games. For instance, the Average Review, Highest Review, and Lowest Review features were expected to be influential as they reflect the ratings displayed on the website when users give their feedback. Additionally, the Game Age and Last Update features provide insight into the game's availability and how long it has been worked on and refined to meet user preferences.

Also, the features closest to the gameplay, namely, the Purchases Count and the Highest Purchase available. These features provide a clear indication of user engagement and interest in the game, and are likely to have a significant impact on user ratings.

Overall, we believe that the features selected for our model provide valuable insight into user behavior in the mobile games market. By focusing on the most relevant features, we can build more accurate and meaningful regression models that help us to understand and predict user behavior, and ultimately create better games that meet the needs and preferences of our target audience.

In conclusion, our team believes that data relevance is a crucial factor in building successful machine learning models. While we have done our best to select the most relevant features for our models, there is always room for improvement in data collection and feature selection. By focusing on the most relevant data, we can create models that are more accurate and meaningful, and that can help us to better understand and predict user behavior in the mobile games market.